IOWA STATE UNIVERSITY

Serialization in Java (Binary and XML)

Kyle Woolcock

ComS 430

4/4/2014

Table of Contents

Introduction

Serialization is the process of converting objects to bytes that can then be used to reconstruct the original object. This process has many applications including remote procedure calls and allowing for persistent data. Serialization is a general programming concept, present in many object-oriented languages. This tutorial will focus on implementations in Java, and how Java handles serialization behind the scenes, but the general concepts can be applied to many languages. Alternatives to certain problems serialization solves to exist. For data persistence, often times databases are used where we just save the information the object stores instead of the object state itself. Java also supports the ability to serialize to XML which is more human readable, and allows for communication between programs written at different times and in different languages. To help read the document, it is important to note that all variable names appear in a different typeset, all class names are bolded, all exceptions are italicized, all method names are followed by parentheses, and all Java keywords appear in bold and italics.

Why Serialize?

Serialization allows for a quick and easy way to store data after a program finishes execution. The serialized data is independent of the Java virtual machine (JVM) that generated it. This means that as long as a different computer has access to the class files and the serialized data, the object can be reconstructed just as it originally was. It also allows for remote procedure calls. To call a method on another machine, often an object is needed argument. Serialization converts an object to a byte stream that can then be sent over a network and deserialized on the target machine.

How to Serialize

Serializable Interface

Java provides two different ways to allow a class to be serialized. The first is to implement the Serializable interface. This is just a marker interface, meaning it contains no methods. Java will also implement a `serialVersionUID` variable, although it is advised manual assign the variable. It is the unique identifier Java uses to tell which class it is reconstructing from a byte stream (more on this later). This is the quickest and easiest way but gives you very little control over how the data is written. Figure 1 shows an implementation of Serializable with an example of a `serialVersionUID` variable. The one in the example was auto generated by Java, but set so that further changes to the class does not affect it. There will be more on this when we talk about versioning in the section on serialization problems.

Externalizable Interface

Externalizable is an interface that extends Serializable. Unlike Serializable, Externalizable is not a marker interface. It requires an implementation of the methods readExternal() and writeExternal(). In addition to these methods, the class must also have a default constructor. This is because when using readExternal() and writeExternal(), a constructor is actually called for the object and then its variables are updates. This allows for a faster execution time than Serializable. To implement readExternal() and writeExternal() manually write the variables of the class to the output stream given. In Figure 2, there is an implementation of the Externalizable Interface along with the readExternal() and writeExternal().

```java
import java.io.Serializable;

public class Rectangle implements Serializable {


    // Optional to specify this, can be auto-generated for you
    private static final long serialVersionUID = -2320627975424434325L;
    int length;
    int width;

    public Rectangle(int length, int width) {
        this.length = length;
        this.width = width;
    }
```

Figure 1

```java
import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;

public class Dog implements Externalizable {

    String name;
    int licenseID;
    Person owner;

    public Dog(String name, int licenseID, Person owner) {

    public Dog() {
    }

    @Override
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
        licenseID = in.readInt();
        name = (String) in.readObject();
        owner = new Person((String) in.readObject(), (String) in.readObject());
    }

    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeInt(licenseID);
        out.writeObject(name);
        out.writeObject(owner.getFirstName());
        out.writeObject(owner.getLastName());
    }

    @Override
    public String toString() {
}
```

Figure 2

Using These Interfaces to Serialize

After implementing Serializable or Externalizable, Java provides two streams to read and write objects: ObjectOutputStream and ObjectInputStream. To create these streams, give them an instance of a file stream that was created with the file to be written to or read from. After doing that, ObjectOutputStream has various methods to write different objects and primitives to the file, notably writeObject(). In Figure 3, there is an example of a main method that is serializing and deserializing an instance of the rectangle class in Figure 1. The yellow highlighting shows the serialization steps whereas the teal highlighting shows the deserialization steps. Most of the exceptions are pretty standard, the only new one of note is *ClassNotFoundException*. This is thrown if the JVM cannot find a class with a `serialVersionUID` matching that of the one read from the file. This can happen if the class files needed to reconstruct the object are not found (either not present on your machine or not in the build path) or if there is a versioning problem, which will be discussed in a later section.

```
public static void main(String[] args) {
    Rectangle rec = new Rectangle(10, 5);
    // Generally use a .ser ending to signal a serialized file
    String filename = "rectangle_example.ser";
    // Serialize the rectangle
    try {
        FileOutputStream fos = new FileOutputStream(filename);
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(rec);
        oos.close();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    // Deserialize the rectangle
    try {
        FileInputStream fis = new FileInputStream(filename);
        ObjectInputStream ois = new ObjectInputStream(fis);
        Rectangle r = (Rectangle) ois.readObject();
        ois.close();
        System.out.println("Length: " + r.length +", Width: " + r.width);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
}
```

Figure 3